An Approach for Mapping Features to Code Based on Static and Dynamic Analysis

Abhishek Rohatgi[1], Abdelwahab Hamou-Lhadj[2], Juergen Rilling[1]
[1]*Department of Computer Science and Software Engineering*
[2]*Department of Electrical and Computer Engineering*
Concordia University
1455 de Maisonneuve West Montreal, Quebec
{a_rohatg, abdelw, rilling@encs.concordia.ca}

Abstract

System evolution depends greatly on the ability of a maintainer to locate source code that is specific to feature implementation. Existing feature location techniques require either exercising several features of the system, or rely heavily on domain experts to guide the feature location process. In this paper, we present a novel approach for feature location that combines static and dynamic analysis techniques. An execution trace is generated by exercising the feature under study (dynamic analysis). A component dependency graph (static analysis) is used to rank the components invoked in the trace according to their relevance to the feature. Our ranking technique is based on the impact of a component modification on the rest of the system. The proposed approach is automatic to a large extent relieving users from any decision that would otherwise require extensive domain knowledge of the system. A case study is presented to support and evaluate the applicability of our approach.

Keywords: Feature location, dynamic analysis, static analysis, program comprehension

1. Introduction

Feature location has been recognised as one of the most important tasks during software evolution and maintenance. Software maintainers typically do not comprehend an entire system to perform a modification request rather they apply an as needed approach, by focusing only on these parts of the source code that is most relevant to the feature or source code to be modified [11].

Existing feature location techniques can be grouped into two main categories depending on their use of static and dynamic analysis techniques. The first category, pure dynamic approaches, require the generation of execution traces that are then compared in order to identify the components of a single feature. An example of these techniques is the one proposed by Wilde and Scully [10], known as Software Reconnaissance. The major limitation with their approach is that it requires execution traces to be generated and processed for many system features, although the focus of a maintenance request is typically only on one particular feature. The second category of feature location techniques relies on a combination of static and dynamic analysis. These approaches utilize static information to further process the execution trace that corresponds to a feature of interest. Several techniques have been presented such as the ones based on concept analysis [2], latent semantic indexing [1], etc. These techniques usually require users to have some understanding of the source code prior to the feature location analysis to be able to specify these parts of the source code that need to be analyzed.

In this paper, we propose a novel technique based on impact analysis for solving the feature location problem. We define a software feature as any specific scenario of a system that is triggered by an external user. This is similar to the concept of scenarios found in UML [13], except that we do not distinguish between primary and exceptional scenarios in this paper. However, it is advisable to include at least the primary scenario, since these scenarios tend to correspond to the most common program execution associated with a particular feature. Our approach combines two different sources of information: an execution trace that corresponds to the software feature under study (dynamic analysis) and a component dependency graph (static analysis). Using the graph, we rank the components invoked in the trace by measuring the impact of a component modification on the rest of the system. Our hypothesis is that the smaller the impact set of a component modification, the more likely it is that the component is specific to a feature. Conversely, we expect a component affecting many parts of a system to be invoked in multiple traces and therefore rendering it as less specific to a particular feature.

The advantage of our approach is that it requires only one trace to be generated, the one corresponding to the feature under study. Furthermore, our approach is almost fully automatic, without requiring users to have an extensive system knowledge prior to performing the analysis.

This work is based on our previous work [9], where we discussed how impact analysis can be applied to address the feature location problem. In this paper, we extend this previous work by introducing two new feature location

techniques based on impact analysis. The two techniques differ in the way the impact of a component modification is measured. We also show results from applying these algorithms to a feature of an object-oriented system.

Organization of the paper: In the next section we describe details of the proposed feature location approach and discuss the application of impact analysis to detect feature components. A case study is presented in Section 3, followed by a discussion on related work in Section 4. Section 5 concludes the paper and present future directions.

2. Approach

Figure 1 provides a general overview of our approach for identifying important components that implement a specific feature. In the context of our research, the components of interest correspond to a system's classes.

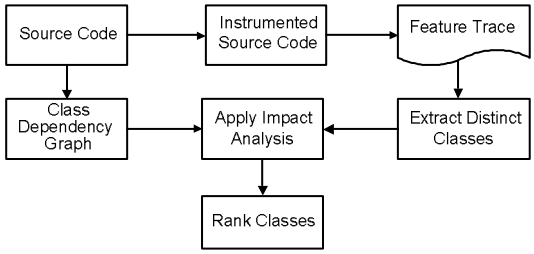

Figure 1. Overall Approach

Our approach is based on the following steps: (1) A trace is generated by exercising the feature to be analyzed and distinct classes are extracted from this trace. (2) We apply our two proposed impact analysis metrics on the extracted classes. Both metrics are based on a static class dependency graph and are discussed in more detail in Section 2.2. (3) Depending on these impact measures, classes are ranked to identify feature specific classes.

2.1. Feature Trace Generation

We first generate a *feature trace* that corresponds to a specific execution of the software feature under study. This step requires the instrumentation of either the source code or the execution environment. The instrumented version is then executed to create a feature trace by exercising the feature of interest. From the feature trace, we can then extract an *execution profile* that corresponds to the distinct classes invoked in the feature trace. It should be noted that the trace does not need to be stored.

2.2. Impact Analysis

Impact analysis is the process of identifying these parts of a program that are potentially affected by a program change. Impact analysis has been shown to be useful for planning changes, making changes, and tracing through the effects of changes [6, 12].

In our approach, we apply impact analysis to identify feature specific classes by measuring the potential impact that modifications to each distinct class in the execution profile has on the remaining parts of a system. The rationale behind this is as follows: classes that impact many other parts of the system will most likely be invoked in many other feature traces, making them non-feature specific. Such high impact classes often correspond to utility classes used to implement core functionality of the system [4]. On the other hand, a feature specific class is self-contained (i.e., low coupling and high cohesion), resulting in a class that when modified has only a very low impact on the remaining parts of the system. In situations where the impact set of a class is in between these two cases, this will indicate classes that implement functionality shared among similar features.

For the measurement of class impact on the remaining parts of a system, we use a static class dependency graph (CDG), which is a directed graph with nodes being a system's classes and edges represent a dependency relationship among these classes (shown in Figure 2).

The construction of a class dependency graph typically requires parsing the source code (or bytecode files). Several types of relationships may exist between two classes such as the ones based on method calls, generalization and realization relationships, etc. It should be noted that the accuracy of the impact analysis depends on the types of dependency relations supported by the analysis, and that edges within a CDG can be weighted to represent the number of dependencies that exist between two given classes.

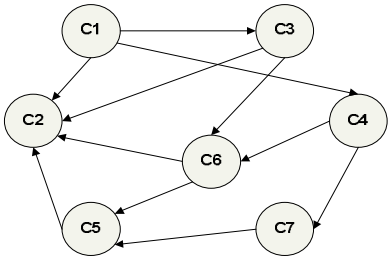

Figure 2. Component Dependency Graph

The impact set of modifying a component *C* is defined as the set of components that depend directly or indirectly on *C*. More formally, a class dependency graph can be represented using a directed graph *G = (V, E)* where *V* is a set of classes and *E* a set of directed edges between classes. The impact set of *C* consists of the set of predecessors of *C*. A predecessor of a node is defined as follows: Consider an edge *e = (x, y)* from node *x* to *y*, If there is a path in the graph that leads from *x* to *y*, then *x* is said to be a predecessor of *y*. For example, the impact set of class *C5* of Figure 2 consists of the classes *C6, C7 C4, C3*, and *C1* (i.e., the predecessors of node *C5*) since there exist a path between each of these classes and the class *C5*. Note that the

same class may occur in multiple paths. In this case, such a class is considered only once.

Based on the definition of the impact set of a component, we have developed two metrics for measuring the impact of a class modification on the rest of the system. In what follows, we introduce the TWI (Two Way Impact) and WTWI (Weighted Two Way Impact) metrics. Before describing these metrics in more detail, we present the following definitions:

- `S` = Set of all classes of the system.
- `P` = Set of all packages of the system.
- `CAI(C)`: Represents the Class Afferent Impact of a class *C*, which consists of the number of classes that are affected (directly or indirectly) when *C* is modified (i.e., the cardinality of the impact set of *C*).
- `CEI(C)`: Represents the Class Efferent Impact of a class *C*, which is the number of classes that will affect (directly or indirectly) *C* if they change. These are the classes in the directed graph that can be reached through *C*. It should be noted that the intersection between CAI and CEI is not necessarily empty, since some components can be affected by a change to *C* while at the same time they can affect *C*.
- `PAI(C)`: Represents the Package Afferent Impact of a class. This metric corresponds to the number of packages affected directly or indirectly by a modification of *C*. We consider all packages of the system as separate packages whether they belong to another package or not.

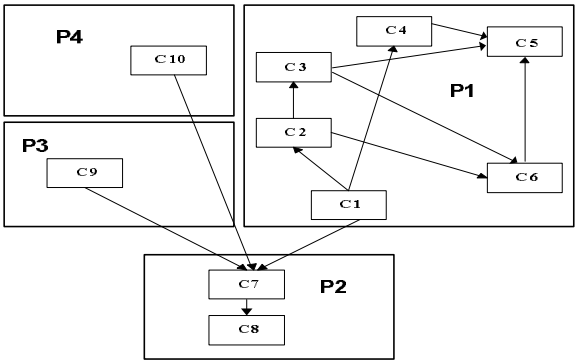

Figure 3. Class dependency graph with packages

We will revisit the example in Figure 3 throughout this section to illustrate how our impact analysis metrics can identify feature related components. In this example, we assume that the classes that are relevant to the specific feature that needs to be analyzed are: *C1, C2,..., C6,* which are all located in package *P1*. However, the feature profile created from this specific feature trace contains additionally the classes *C7* and *C8*.

Metric 1: Two Way Impact (TWI):

The two way impact metric considers both the impact a class modification has on the rest of a system (i.e. the afferent impact), as well as the number of classes that impact this class if these classes change (i.e., efferent impact). We use the CDG to measure the afferent and the efferent impact.

The TWI metric is based on a metric presented by Hamou-Lhadj and Lethbridge in [4]. The authors used fan-in analysis to measure the extent to which a routine can be considered a utility. According to their findings, a routine with high fan-in (incoming edges in the call graph) should be considered a utility as long as its fan-out (outgoing edges) is not high. They argue that a routine that receives a large number of calls from different parts of a program is more likely to be a utility routine. On the other hand, if a routine has many calls (outgoing edges in the call graph), this is evidence that it is performing a complex computation and therefore it is needed to understand the system.

The TWI metric uses a similar approach, except that it considers the impact of a component modification rather than its fan-in. The TWI metric therefore not only considers the direct impact associated with a component change but also the ones that are indirectly affected by this component change. This allows us to measure the fact that the afferent impact of a component can be very high without necessarily having a high fan-in. For example in Figure 4, class *C2* has a very low fan-in (one incoming edge) but a high afferent impact value (five classes are affected by a change to *C2*).

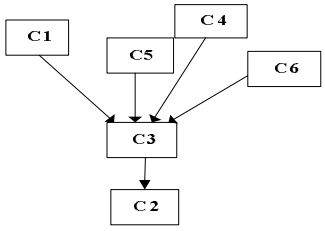

Figure 4. Difference between Fan-in and CAI
Fan-in (C2) = 1 and CAI(C2) = 5

The TWI metric is defined as follows:

$$TWI(c) = \frac{CAI(c)}{|S|} \times \frac{Log(\frac{|S|}{CEI(c)+1})}{Log(|S|)}$$

This formula is divided into two parts. The first part $\frac{CAI(c)}{|S|}$ reflects the fact that the classes with large CAI are the ones that are most likely to be non-feature specific classes, as previously described. We multiply the first part

(i.e. $\frac{CAI(c)}{|S|}$) by a coefficient that takes into account the efferent impact although with a lower emphasis by using the Logarithm. We achieve this using $Log(\frac{|S|}{CEI(c)+1})$. $CEI(c)+1$ is used for convenience to avoid situations where $CEI(c)=0$.

If a class `c` is not affected by any modification made to other classes of the system then $CEI(c)$ equals zero and hence $Log(\frac{|S|}{CEI(c)+1}) = Log(|S|)$. In this case, the result of TWI will depend on the afferent impact.

On the other hand if the efferent impact is considerably large then $Log(\frac{|S|}{CEI(c)+1})$ will converge towards zero and therefore canceling out the effect of $CAI(c)$. This class depends (directly or indirectly) on many other classes and therefore it might be important to understand the implementation of the feature.

We divide the result by $Log(|S|)$ to ensure that both expressions and the entire formula vary from 0 to 1. With 0 being a component that is feature specific and not shared by any component in the system and 1 being a component that is shared among all components in the system.

Table 1. Applying TWI to the example of Figure 3

Classes	CAI	CEI	TWI
C1	**0**	**7**	**0**
C2	**1**	**3**	**0.04**
C4	**1**	**1**	**0.08**
C3	**2**	**2**	**0.12**
C6	**3**	**1**	**0.25**
C7	3	1	0.25
C5	**5**	**0**	**0.63**
C8	5	0	0.63

Table 1 shows the result after applying the TWI metric on the example program (Figure 3). The results are sorted based on the feature specific classes (i.e. TWI value in ascending order). All classes related to the particular feature are clustered together at the top of the table (represented in bold), except `C5` (a feature specific class) that is ranked after `C7` (a non-feature specific class). This is due mainly to the fact that the class afferent impact factor of *`C7`* `(CAI = 3)` is less than *`C5`* `(CAI = 5)`.

Next, we introduce our *Weighted* TWI metrics that improves on our TWI metric by also considering the package afferent impact as part of the measurement.

Metric 2: Weighted Two Way Impact (WTWI):

The WTWI metric uses available information about the system architecture, to weigh the two way impact metric. More specifically, for the WTWI metric we also consider the number of packages affected by a class modification (i.e. the package afferent impact, `PAI`). For example, a class affecting five classes from three different packages is more likely to be part of the execution profile of several features than a class affecting five classes all located in one package.

We introduce the following metric:

$$WTWI(c) = TWI(c) x \frac{PAI(c)}{|P|}$$

The range for the $WTWI(c)$ is from 0 to 1, with 0 being a component that is feature specific and not shared by neither other components or package in the system and 1 being a component that is shared among all components all packages in the system.

Table 2. Applying WTWI to the example of Figure 3

Classes	PAI	WTWI
C1	**1**	**0**
C2	**1**	**0.01**
C4	**1**	**0.02**
C3	**1**	**0.03**
C5	**1**	**0.16**
C6	**1**	**0.06**
C7	4	0.25
C8	4	0.63

Table 2 shows the result after applying the WTWI metric to the example in Figure 3. The results in table 3 are sorted in ascending order based on the feature specific classes (i.e. WTWI value). The table shows that all classes (shown in bold) related to the particular feature are clustered at the top of the table, which improves the results obtained by applying TWI.

3. Case Study

In what follows, we present a case study to evaluate the applicability of our feature location techniques. We apply our techniques on a feature trace generated from a Java-based system called Checkstyle[1] (version 3.3).

Checkstyle is a development tool to support programmers to write Java code that adheres to a coding

[1] http://checkstyle.sourceforge.net/

standard. The tool allows programmers to create XML-based files to represent almost any coding standard. Checkstyle uses ANTLR[2] (ANother Tool for Language Recognition) and the Apache regular expression pattern matching package[3]. These two packages have been excluded from this analysis. Checkstyle (without ANTLR and the Apache module) has 17 packages, 210 classes, and 130 KLOC. The system is well documented and allowing us to validate our results against the documented feature implementations.

3.1. Software Feature

We applied our feature location techniques on a Checkstyle feature called CheckCode, which consists of checking Java code for coding problems such as uninitialized variables, etc. We generated the corresponding feature trace by instrumenting Checkstyle using our own instrumentation tool based on BIT framework [5]. Probes were inserted at each entry and exit method (including constructors). The resulting CheckCode feature trace contained 68 distinct classes.

We applied impact analysis to the Checkstyle systems using a tool called Structural Analysis for Java (SA4J)[4]. The tool parses the source code and generates a global class dependency table that contains various metrics including the class afferent and efferent impacts. SA4J supports a large spectrum of relations among classes such as: accesses, calls, contains, extends, implements, instantiates, references, etc.

3.2. Analysis

The execution profile for the CheckCode feature consists of classes belonging to the following packages: `coding` (32 classes), `checkstyle` (12 classes), `checks` (5 classes), `grammars` (2 classes), and `apis` (17 classes).

We used Checkstyle documentation to identify the most relevant components related to the CheckCode feature. We found that the package `coding` contains the key classes used to implement the various checking procedures relevant to CheckCode feature. Our feature location technique based on the TWI metric did also rank most classes within the `coding` package as important (these classes are not shown in this paper due to limited space). The only major exceptions are the classes: `AbstractSuperCheck` and `AbstractNestedDepthCheck`. Both classes have a large class afferent impact (CAI = 3) factor compared to CAI factor of 1 for the remaining classes in the `coding` package. This is a result of these classes being abstract classes that implement general purpose functions used by many other classes. Therefore, one way to improve our feature location approach is to treat abstract classes separately. We leave out this point as future direction.

The packages `checkstyle`, `checks` and `grammars` invoked in the CheckCode feature trace contain classes that implement common functionality used by most checks performed by Checkstyle. For example the `grammars` package contains operations that build a grammar from the code used in the analysis. Most of these classes in this package are ranked lower than the classes of the `coding` package (with a few exceptions). For example, the class `checkstyle.TreeWalker` was ranked among the most relevant classes, since it provides tree traversal functionalities used by many features within the Checkstyle tool, to navigate the syntax tree created for the Java code to be checked.

The `apis` package is a Checkstyle utility package with most of its classes (except `FilterSet` and `AbstractFileSetCheck`) being correctly ranked low using the TWI based feature location approach.

Applying the WTWI based feature location technique on the CheckCode feature provided noticeably better results than using the TWI metric. Using the WTWI metric, the classes `AbstractSuperCheck` and `AbstractNestedDepthCheck` were identified to be important classes, by significantly closing the gap among these and the other classes of the `coding` package. We also observed an improvement in the ranking of the classes in the `apis` package (closer ranking).

However, the approach placed some classes of the `checks` package such at the classes `AbstractLoader`, `AbstractFormatCheck` (two abstract classes), and `CheckUtils` with the `apis` classes. As mentioned previously, abstract classes seem to behave differently from the other classes of the same packages. As for `CheckUtils`, it is a utility class whose scope is within the `checks` package, unlike the classes of the `apis` package, which are system-level utilities.

4. Related Work

Wilde and Scully [10] introduced the concept of Software Reconnaissance. In their approach, traces are generated by exercising several features. They compared the resulting traces to identify components specific to the feature. In our approach, we only generate a trace for the feature to be analyzed.

Eisenbarth et al. [2] proposed a hybrid feature location approach that uses both static and dynamic analysis techniques. The dynamic information is based on generating traces based on a set of scenarios. Formal concept analysis

[2] http://www.antlr.org/
[3] http://jakarta.apache.org/regexp/
[4] http://www.alphaworks.ibm.com/tech/sa4j

is then applied on the collected execution traces to determine the relation between features. Static analysis is applied to identify additional components relevant to the feature. This approach requires considerable domain knowledge of the software system along with several test cases in order to identify a single feature.

Poshvanyk et al. [7, 8] introduced another hybrid approach based on source code based information retrieval (IR) to describe components invoked in a trace. The advantage of their approach is that it requires only one trace to perform feature location. The disadvantage is that it relies on informal knowledge, such as source code comments, identifiers, etc.

Greevy et al. [3] exploited the relationship between features and classes to analyze the way features of a system evolve and to detect changes in the code from a feature perspective. Rather than detecting feature specific components, the main focus of the authors approach is on studying how the classes may change their roles during software evolution.

5. Conclusion and Future Work

In this paper, we presented a new approach to address the feature location problem with a focus on identifying the classes that are the most relevant to the feature to be analyzed.

Our approach combines both static and dynamic analysis. A trace is generated by exercising a feature under study. The invoked classes in the trace are ranked based on identifying the impact of a class modification on the rest of the system. To measure the impact, a class dependency graph is used. We proposed two impact metrics. The first metric, TWI (Two Way Impact), showed very good results when applied to the trace generated in our case study. It did, however, misplace a few classes (2 out of 32 important classes). The second metric, WTWI (Weighted Two Way Impact), improves the results of TWI by considering information from the system architecture. This can be seen from the results of our case study as WTWI performed well by bringing the two previously misplaced classes closer to the group of important classes.

As part of our future work, we plan to conduct further evaluations of our approach by analyzing larger, less well designed systems. We also plan to apply thresholds to determine automatically then to consider classes to be feature relevant

Acknowledgements

This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, “Indexing by Latent Semantic Analysis”, *Journal of the American Society for Information Science*, *41(6)*, 1990, pp. 391-407.

[2] T. Eisenbarth, R. Koschke, and D. Simon, “Locating Features in Source Code”, *IEEE Transactions on Software Engineering*, *29(3)*, 2003, 210 - 224.

[3] O. Greevy, S. Ducasse, and T. Girba, "Analyzing Feature Traces to Incorporate the Semantics of Change in Software Evolution Analysis", *In Proc. of 21st IEEE Int. Conf. on Software Maintenance*, 2005, pp. 347-356.

[4] A. Hamou-Lhadj, and T. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System", *In Proc. of the 14th IEEE Int. Conf. on Program Comprehension,* 2006, pp. 181-190.

[5] H. Lee, B. G. Zorn, BIT, “A tool for Instrumenting Java Bytecodes”, *USENIX Symposium on Internet technologies and Systems*, 1997, pp. 73-82.

[6] J. Law , G. Rothermel, “Whole program Path-Based dynamic impact analysis”, *In Proc. of the 25th Int. Conf. on Software Engineering,* 2003, pp. 308-318.

[7] D. Poshyvanyk, Y. G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, “Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval”, *IEEE Transactions on Software Engineering, 33(6),* 2007, pp. 420-432.

[8] D. Poshyvanyk, and D. Marcus, “Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code”, *In Proc. of 15th IEEE Int. Conf. on Program Comprehension*, 2007, pp. 37-48.

[9] A. Rohatgi, A. Hamou-Lhadj, J. Rilling, "Feature Location Based on Impact Analysis", *In Proc. of 11th IASTED Int. Conf. on Software Engineering and Applications,* 2007.

[10] N. Wilde, and M. Scully, “Software Reconnaissance: Mapping Program Features to Code”, *Software Maintenance: Research and Practice, 7(1)*, 1995, pp. 49-62.

[11] N. Wilde , M. Buckellew, H. Page , V. Rajlich , L. Pounds, “A comparison of methods for locating features in legacy software”, *Journal of Systems and Software, 65(2)*, 2003, pp.105-114.

[12] S. S. Yau, J. S. Collofello, and T. MacGregor, “Ripple Effect Analysis of Software Maintenance”, *In Proc. of the 4th IEEE Int. Conf. on Computer Software and Applications Conference*, 1978, pp. 60-65.

[13] OMG UML 2.0 Specification: http://www.omg.org/technology/documents/formal/uml.htm